Exercise - 7 : Financial Forecasting Using Recursion

Recursion :

1. Recursion refers to the process of a function calling itself. This technique is mainly used in problems where we have to explore multiple possible paths or for calculating the result of a problem by using the results of it's reduced sub-problems.
2. In our case, we will be forecasting the value of a given principle amount that is invested in compound interest.
3. The code will contain a recursive function that will be used to calculate the value of compound interest gained by the user after investing his principal amount for a given number of years at a predefined rate of interest.
4. Actual formula for calculating return amount after adding the compound interest that can be generated within given time period :

$$A = P\left(1 + \frac{r}{n}\right)^{nt}$$

A: final amount
P: principal deposit
r: annual interest rate
n: number of compounding periods
t: time (expressed in years)

5. Recursive formula for calculating the final amount after adding the compound interest :

$$A(N) = \begin{cases} P & \text{if } N = 0 \\ A(N-1) \cdot (1 + r_p) & \text{if } N > 0 \end{cases}$$

Recursive Code :

```
// Recursive function to compute compound interest
// Here, tp->time period, p->principle amount, r->rate of interest

public static double find_comp(int tp, double p, double r) {

    if (tp == 0) {

        return p;

    }

    return find_comp(tp - 1, principal,r) * (1 + r);

}
```

Complexity Analysis :

Time Complexity : O(n)

- This is because we are making 'n' recursive calls starting 'n' all the way down to '0' (base case)
- Here, 'n' actually refers to the number of time periods we are taking into consideration.

Space Complexity : O(n)

- This is due to stack space being taken up by the 'n' recursive calls that we discussed above i.e., for each recursive call , an activation record will be created and pushed onto the stack thereby making the space complexity O(n).

Improvement :

- We can convert our recursive approach to an iterative one by constructing the solution in a bottom-up way.
- This way, we can eliminate the need to use O(n) stack space by simply using an 'amount' variable to track the changes in each step.
- However, the time complexity will still remain same i.e., O(n)
- But space complexity will become linear i.e., O(1)

Code :

```
package org.example;
import java.util.Scanner;

public class Main {

    // Recursive function to compute compound interest
    public static double find_comp(int tp, double p, double r) {
        if (tp == 0) {
            return p;
        }
        return find_comp(tp - 1, p,r) * (1 + r);
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("--Finacial Forecasting of Compound Interest---");
        // User input
        System.out.print("Enter initial investment (principal) : ");
        double principal = scanner.nextDouble();

        System.out.print("Enter annual interest rate (in %) : ");
        double rate_of_interest = scanner.nextDouble()/100.0;

        System.out.print("Enter number of years for forecasting: ");
        int years = scanner.nextInt();
```

```
        System.out.print("Enter compounding frequency (12 for compounded monthly, 4 for quarterly
etc): ");
        int comps = scanner.nextInt();

        // calculation
        int periods = years * comps;
        double rate_of_interest_per_period = rate_of_interest / comps;

        double return_value = find_comp(periods, principal, rate_of_interest_per_period);

        // output
        System.out.printf("\nForecasted Value after %d years : %.2f\n",
                years, return_value);

        scanner.close();
    }
}
```